```
/*
 * Device driver for the VGA LED Emulator
 *
 * A Platform device implemented using the misc subsystem
 *
 * Stephen A. Edwards
 * Columbia University
 *
 * Modified by: Emily Pakulski (enp2111)
 *
 * References:
 * Linux source: Documentation/driver-model/platform.txt
 *               drivers/misc/arm-charlcd.c
 * http://www.linuxforu.com/tag/linux-device-drivers/
 * http://free-electrons.com/docs/
 *
 * "make" to build
 * insmod rsa_box.ko
 *
 * Check code style with
 * checkpatch.pl --file --no-tree rsa_box.c
 */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>   /* struct file_operations */
#include <linux/uaccess.h>
#include "rsa_box.h"
#include <linux/types.h>

#define DRIVER_NAME "rsa_box"

/*
 * Information about our device
 */
struct vga_led_dev {
        struct resource res; /* Resource: our registers */
        void __iomem *virtbase; /* Where registers can be accessed in memory */
} dev;

/*
 * Write segments of a single digit
 * Assumes digit is in range and the device information has been set up
 */
static void write_digit(int address, u32 segments)
{
        iowrite32(segments, dev.virtbase + address * 4);
}

static u32 read_digit(int address)
{
        u32 answer;
        answer = ioread32(dev.virtbase + address * 4);
        return answer;
```

```
 64: }
 65:
 66: /*
 67:  * Handle ioctl() calls from userspace:
 68:  * Read or write the segments on single digits.
 69:  * Note extensive error checking of arguments
 70:  */
 71: static long vga_led_ioctl(struct file *f, uint32_t cmd, unsigned long arg)
 72: {
 73:         rsa_box_arg_t vla;
 74:
 75:         switch (cmd) {
 76:         case RSA_BOX_WRITE_DIGIT:
 77:                 if (copy_from_user(&vla, (rsa_box_arg_t *) arg,
 78:                                    sizeof(rsa_box_arg_t)))
 79:                         return -EACCES;
 80:                 write_digit(vla.address, vla.data_in);
 81:                 break;
 82:
 83:         case RSA_BOX_READ_DIGIT:
 84:                 if (copy_from_user(&vla, (rsa_box_arg_t *) arg,
 85:                                    sizeof(rsa_box_arg_t)))
 86:                         return -EACCES;
 87:                 vla.data_in = read_digit(vla.address);
 88:                 if (copy_to_user((rsa_box_arg_t *) arg, &vla,
 89:                                  sizeof(rsa_box_arg_t)))
 90:                         return -EACCES;
 91:                 break;
 92:
 93:         default:
 94:                 return -EINVAL;
 95:         }
 96:
 97:         return 0;
 98: }
 99:
100: /* The operations our device knows how to do */
101: // www.tdlp.org/LDP/lkmpg/2.4/html/c577.htm
102: static const struct file_operations vga_led_fops = {
103:         .owner          = THIS_MODULE,
104:         .unlocked_ioctl = vga_led_ioctl,
105: };
106:
107: // file_operations holds pointers to functions defined by the driver that performs
108: // various operations on the device.
109: // Each field of the structure corresponds to the address of some function
110: // defined by the driver to handle a requested operation.
111:
112: /* Information about our device for the "misc" framework -- like a char dev */
113: static struct miscdevice vga_led_misc_device = {
114:         .minor          = MISC_DYNAMIC_MINOR,
115:         .name           = DRIVER_NAME,
116:         .fops           = &vga_led_fops,
117: };
118:
119: /*
120:  * Initialization code: get resources (registers) and display
121:  * a welcome message
122:  */
123: static int __init vga_led_probe(struct platform_device *pdev)
124: {
125:         int ret;
126:
```

```
127:         /* Register ourselves as a misc device: creates /dev/rsa_box */
128:         ret = misc_register(&vga_led_misc_device);
129:
130:         /* Get the address of our registers from the device tree */
131:         ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
132:         if (ret) {
133:                 ret = -ENOENT;
134:                 goto out_deregister;
135:         }
136:
137:         /* Make sure we can use these registers */
138:         if (request_mem_region(dev.res.start, resource_size(&dev.res),
139:                                DRIVER_NAME) == NULL) {
140:                 ret = -EBUSY;
141:                 goto out_deregister;
142:         }
143:
144:         /* Arrange access to our registers */
145:         dev.virtbase = of_iomap(pdev->dev.of_node, 0);
146:         if (dev.virtbase == NULL) {
147:                 ret = -ENOMEM;
148:                 goto out_release_mem_region;
149:         }
150:
151:         return 0;
152:
153: out_release_mem_region:
154:         release_mem_region(dev.res.start, resource_size(&dev.res));
155: out_deregister:
156:         misc_deregister(&vga_led_misc_device);
157:         return ret;
158: }
159:
160: /* Clean-up code: release resources */
161: static int vga_led_remove(struct platform_device *pdev)
162: {
163:         iounmap(dev.virtbase);
164:         release_mem_region(dev.res.start, resource_size(&dev.res));
165:         misc_deregister(&vga_led_misc_device);
166:         return 0;
167: }
168:
169: /* Which "compatible" string(s) to search for in the Device Tree */
170: #ifdef CONFIG_OF
171: static const struct of_device_id vga_led_of_match[] = {
172:         { .compatible = "altr,rsa_box" },
173:         {},
174: };
175: MODULE_DEVICE_TABLE(of, vga_led_of_match);
176: #endif
177:
178: /* Information for registering ourselves as a "platform" driver */
179: static struct platform_driver vga_led_driver = {
180:         .driver = {
181:                 .name   = DRIVER_NAME,
182:                 .owner  = THIS_MODULE,
183:                 .of_match_table = of_match_ptr(vga_led_of_match),
184:         },
185:         .remove = __exit_p(vga_led_remove),
186: };
187:
188: /* Called when the module is loaded: set things up */
189: static int __init vga_led_init(void)
```

```
{
        pr_info(DRIVER_NAME ": init\n");
        return platform_driver_probe(&vga_led_driver, vga_led_probe);
}

/* Called when the module is unloaded: release resources */
static void __exit vga_led_exit(void)
{
        platform_driver_unregister(&vga_led_driver);
        pr_info(DRIVER_NAME ": exit\n");
}

module_init(vga_led_init);
module_exit(vga_led_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("RSA Box Team - Columbia University (based on code by Professor Stephen Edwards at Columbia)");
MODULE_DESCRIPTION("RSA Box - hardware RSA implementation device driver");
```

```
 1: #ifndef _RSA_BOX_H
 2: #define _RSA_BOX_H
 3:
 4: #include <linux/ioctl.h>
 5: #include <linux/types.h>     /* for int32_t */
 6:
 7: typedef struct {
 8:   int address;
 9:   uint32_t data_in;
10: } rsa_box_arg_t;
11:
12: #define RSA_BOX_MAGIC 'q'
13:
14: /* ioctls and their arguments */
15: #define RSA_BOX_WRITE_DIGIT _IOW(RSA_BOX_MAGIC, 1, rsa_box_arg_t *)
16: #define RSA_BOX_READ_DIGIT  _IOWR(RSA_BOX_MAGIC, 2, rsa_box_arg_t *)
17:
18: #endif
```

```
    1: /* verilator lint_off WIDTH */
    2: /* verilator lint_off UNUSED */
    3: /*
    4: * Code for RSA Box, a hardware implementation of the RSA algorithm.
    5:  */
    6:
    7:  module RSA_BOX(input logic      clk,
    8:      input logic             reset,
    9:      input logic             write,
   10:      input                   chipselect,
   11:      input logic[31:0]       data_in, // the current 32 bit input
   12:      input logic[2:0]        address, // which 32 bit segment of each structure to writ
e into
   13:      output logic[31:0]      data_out
   14: //                output        logic                             is_ready
   15:     );
   16:
   17:
   18:     /* instruction bits (can pick from instructions defined in user-level/instructions.
h) */
   19:     logic[31:0] instrBits;
   20:     /* structures/registers */
   21:     logic[127:0] outputBits;
   22:     // private keys
   23:     logic[63:0] p;
   24:     logic[63:0] q;
   25:     logic[127:0] m;
   26:     // public keys
   27:     logic [127:0] c;
   28:     logic[127:0]        n; // p * q
   29:     logic[31:0] e;
   30:     /* enabler for ALU */
   31:     logic[1:0] functionCall;
   32:     logic ready;
   33:     logic ready_for_encrypt;
   34:     logic reset_exponent;
   35:
   36:     logic reset_decrypt;
   37:     logic[127:0] m1;
   38:     logic [127:0] d;
   39:     logic[127:0] decrypt_message;
   40:     logic ready_for_decrypt;
   41:          logic ready_for_read;
   42:     logic[127:0] our_n;
   43:          //assign reset_exponent = (reset | reset_exponent_signal);
   44:
   45:          exponentiate encryptModule(.reset(reset_exponent), .clk(clk), .m(m), .e(e), .n
(n), .c(c), .ready(ready_for_encrypt)) ;
   46:          exponentiate decryptModule(.reset(reset_decrypt), .clk(clk), .m(m1), .e(d), .n
(our_n), .c(decrypt_message), .ready(ready_for_decrypt)) ;
   47:
   48:          always_ff @(posedge clk) begin
   49:              if (reset || (address == 3'b000 && instrBits == 1'b1)) begin
   50:                  /* reset triggered when clock starts */
   51:                  data_out[31:0] <=             32'd0;
   52:                  instrBits[31:0] <=            32'd0;     // reset typeof(instr)
   53:                  p[63:0] <=                                    64'd0;
   54:                  q[63:0] <=                                    64'd0;
   55:                  n[127:0] <=                                   128'd0;
   56:                  e[31:0] <=                                    32'd0;
   57:                  m[127:0] <= 128'd0;
   58:                  ready <= 1'b0;
   59:                  d[127:0] <= 128'd0;
```

```
   60:                 m1[127:0] <= 128'd0;
   61:                 functionCall <= 2'b00;
   62:                 reset_exponent <= 1'b1;
   63:                 reset_decrypt <= 1'b1;
   64:                 our_n[127:0] <= 128'd0;
   65:             end
   66:
   67:             /* reading */
   68:             if (chipselect && !write) begin
   69:                 case(functionCall)
   70:                     2'b01: begin //encrypt
   71:                         case (address)
   72:                             3'b000: data_out[31:0] <= c[31:0];
   73:                             3'b001: data_out[31:0] <= c[63:32];
   74:                             3'b010: data_out[31:0] <= c[95:64];
   75:                             3'b011: data_out[31:0] <= c[127:96];
   76:                             3'b100: data_out[0] <= ready_for_encrypt;
   77:                             default: begin end
   78:                         endcase
   79:                     end
   80:                     2'b10: begin
   81:                         case (address) //decrypt
   82:                             3'b000: data_out[31:0] <= decrypt_message[31:0];
   83:                             3'b001: data_out[31:0] <= decrypt_message[63:32];
   84:                             3'b010: data_out[31:0] <= decrypt_message[95:64];
   85:                             3'b011: data_out[31:0] <= decrypt_message[127:96];
   86:                             3'b100: data_out[1] <= ready_for_decrypt;
   87:                             default: begin end
   88:                         endcase
   89:                     end
   90:                     2'b11: begin //multiply to read from n or n1 (depending on what's
on output bits)
   91:                                                                                 case (a
ddress)
   92:                             3'b000: data_out[31:0] <= outputBits[31:0];
   93:                             3'b001: data_out[31:0] <= outputBits[63:32];
   94:                             3'b010: data_out[31:0] <= outputBits[95:64];
   95:                             3'b011: data_out[31:0] <= outputBits[127:96];
   96:                             3'b100: data_out[1] <= ready_for_read;
   97:                             default: begin end
   98:                         endcase
   99:                     end
  100:                     default: begin end
  101:                 endcase
  102:             end
  103:             /* writing */
  104:             else if (chipselect && write) begin
  105:                 /* determine what kind of instruction this is */
  106:                 if (address == 3'b000) begin
  107:                     instrBits[31:0] <= data_in[31:0];
  108:                 end
  109:
  110:                 /****** INSTRUCTIONS: check which each instruction *******/
  111:                 /* STORE_PUBLIC_KEY_1: n */
  112:                 case(instrBits)
  113:                     32'd2: begin
  114:                         case(address)
  115:                             3'b001: n[31:0] <=         data_in[31:0];
  116:                             3'b010: n[63:32] <=        data_in[31:0];
  117:                             3'b011: n[95:64] <=        data_in[31:0];
  118:                             3'b100: n[127:96] <= data_in[31:0];
  119:                             default: begin end
  120:                         endcase
```

```
121:                        end
122:                        32'd3: begin
123:                            /* STORE_PUBLIC_KEY_2: e */
124:                            case(address)
125:                                3'b001: begin
126:                                    e[31:0] <=     data_in[31:0];
127:                                end
128:                                default: begin end
129:                            endcase
130:                        end
131:                        32'd4: begin
132:                            /* STORE_PRIVATE_KEY_1: p */
133:                            case(address)
134:                                3'b001: p[31:0] <=         data_in[31:0];
135:                                3'b010: p[63:32] <=        data_in[31:0];
136:                                default: begin end
137:                            endcase
138:                        end
139:                        32'd5: begin
140:                            /* STORE_PRIVATE_KEY_2: q */
141:                            case(address)
142:                                3'b001: q[31:0] <= data_in[31:0];
143:                                3'b010: q[63:32] <= data_in[31:0];
144:                                default: begin end
145:                            endcase
146:                        end
147:
148:                        32'd6: begin
149:                            /* DECRYPT_BITS */
150:                            case(address)
151:                                3'b001: begin
152:                                    functionCall[1:0] <= 2'b10;
153:                                    reset_decrypt <= 1'b1;
154:                                end
155:                                3'b010: begin
156:                                    functionCall[1:0] <= 2'b10;
157:                                    reset_decrypt <= 1'b0;
158:                                end
159:                                default: begin end
160:                            endcase
161:                        end
162:
163:                        32'd7: begin
164:                            /* ENCRYPT_BITS */
165:                            case(address)
166:                                3'b001: begin
167:                                    functionCall[1:0] <= 2'b01;
168:                                    reset_exponent <= 1'b1;
169:                                end
170:                                3'b010: begin
171:                                    functionCall[1:0] <= 2'b01;
172:                                    reset_exponent <= 1'b0;
173:                                end
174:                                default: begin end
175:                            endcase
176:                        end
177:                        32'd8: begin
178:                            /* READ_PUBLIC_KEY_1: n */
179:                            case (address)
180:                                3'b001: begin
181:
      outputBits[127:0] <=    n[127:0];
182:
```

```
      ready_for_read <= 1'b1;
183:
      functionCall <= 2'b11;
184:                                                                    end
185:                             default: begin end
186:                         endcase
187:                     end
188:                     32'd9: begin
189:                         /* READ_PUBLIC_KEY_2: e */
190:                         case(address)
191:                             3'b001: begin
192:                                 outputBits[31:0] <= e[31:0];
193:                             end
194:                             default: begin end
195:                         endcase
196:                     end
197:                     32'd10: begin
198:                         /* STORE_MESSAGE: m*/
199:                         case (address)
200:                             3'b001: m[31:0] <=         data_in[31:0];
201:                             3'b010: m[63:32] <=        data_in[31:0];
202:                             3'b011: m[95:64] <=        data_in[31:0];
203:                             3'b100: m[127:96] <=       data_in[31:0];
204:                             default: begin end
205:                         endcase
206:                     end
207:
208:                     32'd11: begin
209:                         /* STORE_MESSAGE: m1*/
210:                         case (address)
211:                             3'b001: m1[31:0] <=        data_in[31:0];
212:                             3'b010: m1[63:32] <=       data_in[31:0];
213:                             3'b011: m1[95:64] <=       data_in[31:0];
214:                             3'b100: m1[127:96] <=      data_in[31:0];
215:                             default: begin end
216:                         endcase
217:                     end
218:
219:
220:                     32'd12: begin
221:                         /* STORE D*/
222:                         case (address)
223:                             3'b001: d[31:0] <=         data_in[31:0];
224:                             3'b010: d[63:32] <=        data_in[31:0];
225:                             3'b011: d[95:64] <=        data_in[31:0];
226:                             3'b100: d[127:96] <=       data_in[31:0];
227:                             default: begin end
228:                         endcase
229:                     end
230:
231:
232:                     32'd13: begin
233:                         /* STORE D*/
234:                         case (address)
235:                             3'b001: our_n[127:0] <= p[63:0] * q[63:0];
236:                             default: begin end
237:                         endcase
238:                     end
239:
240:                                                 32'd14: begin
241:                         /* READ_PUBLIC_KEY_1: n */
242:                         case (address)
243:                             3'b001: begin
```

```
  244:                                                                 outputBits[12
7:0] <=         our_n[127:0];
  245:                                                                 ready_for_rea
d <= 1'b1;
  246:                                                                 functionCall
<= 2'b11;
  247:                                                                 end
  248:                                 default: begin end
  249:                             endcase
  250:                         end
  251:
  252:                         default: begin
  253:                         end
  254:                     endcase
  255:                 end // end for _writing_
  256:             end // end always_ff
  257:             endmodule
  258:
  259:
  260:
  261:
  262: module exponentiate( input logic reset, clk,
  263:     input logic[127:0] m,
  264:     input logic[127:0] e,
  265:     input logic[127:0] n,
  266:     output logic[127:0] c,
  267:     output logic ready
  268: );
  269:
  270: logic[127:0] base;
  271: logic mult_ready;
  272: logic square_ready;
  273: logic fun;
  274: logic[127:0] squared;
  275: logic[127:0] product;
  276: logic mult_reset;
  277: logic square_reset;
  278: logic new_mult;
  279: logic new_square;
  280: logic[127:0] temp;
  281: logic[127:0] exp;
  282:
  283: incrementA multiply(
  284:     .reset      (mult_reset),
  285:     .clk,
  286:     .a          (base),
  287:     .b          (c),
  288:     .outputAnswer (product),
  289:     .ready      (mult_ready),
  290:     .n
  291: );
  292:
  293: incrementA square(
  294:     .reset      (square_reset),
  295:     .clk,
  296:     .a          (base),
  297:     .b          (base),
  298:     .outputAnswer (squared),
  299:     .ready      (square_ready),
  300:     .n
  301: );
  302:
  303: assign mult_reset = (reset | new_mult);
```

```
304: assign square_reset = (reset | new_square);
305:
306: always_ff @(posedge clk)
307: begin
308:     if(reset)
309:     begin
310:         ready <= 0;
311:         c <= 128'd1;
312:         base <= m;
313:         fun <= 1'b0;
314:         new_mult <= 0;
315:         new_square <= 0;
316:         exp <= e;
317:     end
318:     else if(exp > 32'b0) begin
319:         case(fun)
320:             1'b0: begin
321:                 new_mult <= 1;
322:                 new_square <= 1;
323:                 if(!mult_ready & !square_ready)
324:                     fun <= 1'b1;
325:             end
326:             1'b1: begin
327:                 new_mult <= 0;
328:                 new_square <= 0;
329:                 if(mult_ready & square_ready) begin
330:                     if(exp[0])
331:                         c <= product;
332:                     base <= squared;
333:                     fun <= 1'b0;
334:                     exp <= exp >> 1;
335:                 end
336:             end
337:         endcase
338:     end
339:     else
340:         ready <= 1;
341: end
342: endmodule
343:
344:
345:
346:
347: /* verilator lint_off UNUSED */
348: /* verilator lint_off WIDTH */
349: /* verilator lint_off UNSIGNED */
350: module incrementA(input logic reset,
351:     input logic clk,
352:     input logic[127:0] a,
353:     input logic[127:0] b,
354:     input logic[127:0] n,
355:     output logic ready,
356:     output logic[127:0] outputAnswer
357: );
358:
359: logic[8:0] counter;
360:
361: logic[127:0] a_and_zero;
362: logic[127:0] not_a_and_zero;
363: logic[127:0] a_and_n;
364: logic[127:0] not_a_and_n;
365:
366: logic[127:0] a_and_two_n;
```

```
  367: logic[127:0] not_a_and_two_n;
  368: logic[127:0] twoN;
  369: logic fun;
  370:
  371: logic[127:0] r;
  372: logic[127:0] twoR;
  373: logic[127:0] b_minus_n;
  374: logic[127:0] b_minus_two_n;
  375:
  376: logic out;
  377:
  378: always_ff @(posedge clk)
  379: begin
  380:     if(reset)
  381:     begin
  382:         ready <= 0;
  383:         counter[8:0] <= 9'd128;
  384:
  385:         r[127:0] <= 6'd0;
  386:         a_and_zero[127:0] <= 9'd0;
  387:         not_a_and_zero[127:0] <= 9'd0;
  388:
  389:         a_and_n[127:0] <= 9'd0;
  390:         not_a_and_n[127:0] <= 9'd0;
  391:
  392:         a_and_two_n[127:0] <= 9'd0;
  393:         not_a_and_two_n[127:0] <= 9'd0;
  394:
  395:         twoN[127:0] <= n[127:0]<<1;
  396:         fun <= 1'b1;
  397:
  398:         b_minus_n[127:0] <= b[127:0] - n[127:0];
  399:
  400:     end
  401:     else
  402:     begin
  403:         b_minus_two_n[127:0] <= b[127:0] - twoN[127:0];
  404:         case(fun)
  405:             1'b0: begin
  406:                 if($signed(counter) == -9'd1) begin
  407:                     outputAnswer[127:0] <= r[127:0];
  408:                     ready <= 1'b1;
  409:                 end
  410:                 else begin
  411:                     out <= a[counter];
  412:                     a_and_zero[127:0] <= (twoR[127:0] + b[127:0] );
  413:                     not_a_and_zero[127:0] <= (twoR[127:0]);
  414:                     a_and_n[127:0] <= (twoR[127:0] + b_minus_n[127:0]);
  415:                     not_a_and_n[127:0] <= (twoR[127:0] - n[127:0]);
  416:                     a_and_two_n[127:0] <= (twoR[127:0] + b_minus_two_n[127:0]);
  417:                     not_a_and_two_n[127:0] <= (twoR[127:0] - twoN[127:0]);
  418:                     fun <= 1'b1;
  419:                 end
  420:             end
  421:             1'b1: begin
  422:                 if($signed(counter) >= 9'd0) begin
  423:                     counter <= $signed(counter) - 1'b1;
  424:                     fun <= 1'b0;
  425:                     case(out)
  426:                         1'b0: begin
  427:                             if($signed(not_a_and_zero[127:0]) >= 0 && not_a_and_zero[12
7:0]<n[127:0]) begin
  428:                                 r[127:0] <= not_a_and_zero[127:0];
```

```
  429:                                 twoR[127:0] <= not_a_and_zero[127:0] <<1;
  430:                             end
  431:                             else if($signed(not_a_and_n[127:0]) >= 0 && not_a_and_n[127
:0]<n[127:0]) begin
  432:                                 r[127:0] <= not_a_and_n[127:0];
  433:                                 twoR[127:0] <= not_a_and_n[127:0] <<1;
  434:                             end
  435:                             else begin
  436:                                 r[127:0] <= not_a_and_two_n[127:0];
  437:                                 twoR[127:0] <= not_a_and_two_n[127:0] <<1;
  438:                             end
  439:                         end
  440:                         1'b1: begin
  441:                             if($signed(a_and_zero[127:0]) >= 0 && a_and_zero[127:0]<n[1
27:0]) begin
  442:                                 r[127:0] <= a_and_zero[127:0];
  443:                                 twoR[127:0] <= a_and_zero[127:0] <<1;
  444:                             end
  445:                             else if($signed(a_and_n[127:0]) >= 0 && a_and_n[127:0]<n[12
7:0]) begin
  446:                                 r[127:0] <= a_and_n[127:0];
  447:                                 twoR[127:0] <= a_and_n[127:0] <<1;
  448:                             end
  449:                             else begin
  450:                                 r[127:0] <= a_and_two_n[127:0];
  451:                                 twoR[127:0] <= a_and_two_n[127:0] <<1;
  452:                             end
  453:                         end
  454:                     endcase
  455:                 end
  456:
  457:             end
  458:
  459:         endcase
  460:     end
  461: end
  462: endmodule
  463:
  464:
  465:
```

```
 1: #ifndef __C_INTERFACE_H__
 2: #define __C_INTERFACE_H__
 3:
 4: #include <stdint.h>     /* for unit32_t */
 5:
 6: /*
 7:  * Set private keys to allow encrypting. Set public keys
 8:  * to allow decrypting.
 9:  */
10: void set_private_keys(int32_t *p, int32_t *q);
11: void set_public_keys(int32_t *e, int32_t *n);
12: void __read_public_keys(int32_t *e, int32_t *n);
13:
14: // Encryption and decryption using values stored in registers.
15: // Raise exception and set errno if relevant register not set.
16: void encrypt(char *msg_buf, int32_t *cypher_buf, int len);
17: void decrypt(int32_t *cypher_buf, char *msg_buf, int len);
18:
19: #endif
```

```
#ifndef __C_WRAPPER_H__
#define __C_WRAPPER_H__

#define PRIVATE                 0
#define PUBLIC                  1
#define DECRYPT_SEND            0
#define ENCRYPT_SEND            1

// comment or uncommon line 10 to add/remove debug print statements
#define PRINTVERBOSE            1

/* store private keys, getting back public key */
void key_swap(int32_t *p, int32_t *q, int32_t *our_n);

/* encrypt or decrypt */
void send_int_encrypt_decrypt(int action, int32_t *message_n, int32_t *output);

/* read back value encrypted/decrypted */
void __read_encryption(int32_t *encryption);
void __read_decryption(int32_t *decryption);
void read_our_N(int32_t *n);

/* helper functions */
void set_fd();
void print_128_bit_integer(int32_t *input_x);

#endif
```

```
1: #ifndef __EXTEUC_H__
2: #define __EXTEUC_H__
3:
4: #include <stdint.h>     /* for unit32_t */
5:
6: void err_sys(char *err);
7: void e_euclid(int32_t e, int32_t phi[4], int32_t *d);
8:
9: #endif
```

```
#ifndef __INSTRUCTIONS_H__
#define __INSTRUCTIONS_H__

/*
 * before writing any data, specify which instruction will be used:
 * write INSTRUCTION with desired action (e.g. MAKE_KEY, ENCRYPT, etc)
 */
#define INSTRUCTION             0
#define RESET                   1
#define STORE_PUBLIC_KEY_1      2 // n
#define STORE_PUBLIC_KEY_2      3 // e
#define STORE_PRIVATE_KEY_1     4 // p
#define STORE_PRIVATE_KEY_2     5 // q
#define DECRYPT_BITS            6 // DECRYPT_3
#define ENCRYPT_BITS            7
#define READ_PUBLIC_KEY_1       8 // n
#define READ_PUBLIC_KEY_2       9 // e
#define STORE_MESSAGE           10 // m
#define STORE_MESSAGE2          11 // m
#define STORE_D                 12 // m
#define MAKE_OUR_N              13 // carry out p * q op
#define READ_OUR_N              14 // read back (p * q)

void log_instruction(int opcode);

#endif
```

```
 1: #ifndef _PRIMEGENERATOR_H_
 2: #define _PRIMEGENERATOR_H_
 3:
 4: #include <stdint.h>
 5: #include <inttypes.h>
 6:
 7: /* GNU C seeder */
 8: unsigned long long rdtsc();
 9: /* modular exponentiation */
10: uint64_t modulo(uint64_t base, uint64_t exponent, uint64_t mod);
11: /* Miller-Rabin Primality Test */
12: int miller(uint64_t p, int iteration);
13: uint64_t get_random(int tries);
14: uint64_t generate_prime();
15: void generate_prime_as_int32_t(int32_t *prime_64);
16:
17: #endif
```

```c
#include <stdint.h>             /* for unit32_t */
#include <stdlib.h>             /* for malloc */
#include <stdio.h>              /* for printf */
#include <string.h>             /* for memcpy */
#include "c-interface.h"
#include "c-wrapper.h"          /* for all functions making syscalls */

#define TRUE    1
#define FALSE   0

void set_private_keys(int32_t *p, int32_t *q)
{
    store_keys(PRIVATE, p, q);
}

void set_public_keys(int32_t *e, int32_t *n)
{
    store_keys(PUBLIC, e, n);
}

void read_public_keys(int32_t *e, int32_t *n)
{
    __read_public_keys(e, n);
}

/*
 * encrypt message and return as 32-bit int array.
 */
void encrypt(char *msg_buf, int32_t *cypher_buf, int len)
{
    int i;
    int32_t curr_val;

    for (i = 0; i < len; i++)
    {
        memcpy(&curr_val, msg_buf + i, sizeof(int32_t));
       // send_int_encrypt_decrypt(ENCRYPT_SEND, &curr_val);
        memcpy(cypher_buf + i, &curr_val, sizeof(char));
    }
}

/*
 * decrypt cypher and return message as char array.
 */
void decrypt(int32_t *cypher_buf, char *msg_buf, int len)
{
    int i;
    int32_t curr_val;

    for (i = 0; i < len; i++)
    {
        memcpy(&curr_val, cypher_buf + i, sizeof(int32_t));
       // send_int_encrypt_decrypt(DECRYPT_SEND, &curr_val);
        memcpy(msg_buf + i, &curr_val, sizeof(char));
    }
}
```

```
 1: /*
 2:  * Userspace program that communicates with the RSA_Box device driver
 3:  * primarily through ioctls.
 4:  *
 5:  * Original VGA_LED code by Stephen A. Edwards, Columbia University
 6:  */
 7:
 8: #include <stdio.h>
 9: #include <unistd.h>
10: #include <stdlib.h>
11: #include <sys/ioctl.h>
12: #include <sys/types.h>
13: #include <sys/stat.h>
14: #include <fcntl.h>
15: #include <string.h>
16: #include <time.h>       /* for sleep() */
17: #include <stdint.h>     /* for unit32_t */
18: #include "../rsa_box.h"
19: #include "instructions.h"
20: #include "c-wrapper.h"
21: #include "exteuc.h"
22:
23: void read_segment(int32_t *bit_output, int size);
24: void send_bits(int32_t *value, int count);
25: void __store_d(int32_t *d);
26: void store_keys(int type, int32_t *key_1, int32_t *key_2);
27:
28: /* globals */
29: static int BIT_SEGMENTS[5] =  {1, 2, 3, 4, 5};
30: static int BIT_SEGMENTS_READ[5] = {0, 1, 2, 3, 4};
31: static int rsa_box_fd = -1;
32: static int empty[4] = {0, 0, 0, 0};
33:
34: void set_fd()
35: {
36:     char *filename = "/dev/rsa_box";
37:     if ( (rsa_box_fd = open(filename, O_RDWR)) == -1)
38:     {
39:         fprintf(stderr, "could not open %s\n", filename);
40:     }
41: }
42:
43: /*
44:  * Tells hardware what instruction to include the incoming
45:  * data with.
46:  */
47: void send_instruction(int operation)
48: {
49:     rsa_box_arg_t rsa_userspace_vals;
50:     if (rsa_box_fd == -1)
51:         set_fd();
52:
53:     rsa_userspace_vals.address =  INSTRUCTION;
54:     rsa_userspace_vals.data_in = operation;
55:
56: #ifdef PRINTVERBOSE
57:     log_instruction(operation);
58: #endif
59:
60:     if (ioctl(rsa_box_fd, RSA_BOX_WRITE_DIGIT, &rsa_userspace_vals))
61:     {
62:         perror("ioctl(RSA_BOX_WRITE_DIGIT) failed");
63:     }
```

```
64: }
65:
66: /*
67:  * Sends count int32_t's to the hardware.
68:  * Always call send_instruction() first or the hardware won't know
69:  * what to do with the incoming data.
70:  */
71: void send_bits(int32_t *value, int count)
72: {
73:     rsa_box_arg_t rsa_userspace_vals;
74:     int i;
75:
76:     if (rsa_box_fd == -1)
77:         set_fd();
78:
79:     for (i = 0; i < count; i++)
80:     {
81:         rsa_userspace_vals.address = BIT_SEGMENTS[i];
82:         rsa_userspace_vals.data_in = value[i];
83:
84: #ifdef PRINTVERBOSE
85:         printf("[sending] %d // %d\n", BIT_SEGMENTS[i], value[i]);
86: #endif
87:
88:         if (ioctl(rsa_box_fd, RSA_BOX_WRITE_DIGIT, &rsa_userspace_vals))
89:         {
90:             perror("ioctl(RSA_BOX_WRITE_DIGIT) failed");
91:         }
92:     }
93:
94: }
95:
96: /*
97:  * Store private keys and get back our public key.
98:  */
99: void key_swap(int32_t *p, int32_t *q, int32_t *our_n)
100: {
101:     int32_t p_phi[2];
102:     int32_t q_phi[2];
103:     int32_t phi_n[4];
104:     int32_t d[4];
105:
106:     // calculate p - 1, q - 1
107:     p_phi[0] = p[0] - 1;
108:     p_phi[1] = p[1];
109:
110:     q_phi[0] = q[0] - 1;
111:     q_phi[1] = q[1];
112:
113:     // store d, the extended euclid of (p - 1)(q - 1) and e
114:     store_keys(PRIVATE, p_phi, q_phi);
115:     read_our_N(phi_n);
116:
117:     int32_t E = 65537;
118:     e_euclid(E, phi_n, d);
119:     __store_d(d);
120:
121:     // store actual p and q
122:     store_keys(PRIVATE, p, q);
123:     read_our_N(our_n);
124: }
125:
126: /*
```

```
 127:  * Stores keys into the specified registers, PUBLIC or PRIVATE
 128:  * key registers.
 129:  */
 130: void store_keys(int type, int32_t *key_1, int32_t *key_2)
 131: {
 132:     if (type == PRIVATE)
 133:     {
 134:         send_instruction(STORE_PRIVATE_KEY_1);
 135:         send_bits(key_1, 2); // p
 136:         send_instruction(STORE_PRIVATE_KEY_2);
 137:         send_bits(key_2, 2); // q
 138:     }
 139:
 140:     if (type == PUBLIC)
 141:     {
 142:         send_instruction(STORE_PUBLIC_KEY_1);
 143:         send_bits(key_1, 4); // n
 144:         send_instruction(STORE_PUBLIC_KEY_2);
 145:         send_bits(key_2, 1); // e
 146:     }
 147: }
 148:
 149:
 150: void __store_d(int32_t *d)
 151: {
 152:     send_instruction(STORE_D);
 153:     send_bits(d, 4);
 154: }
 155:
 156: /*
 157:  * Writes input to m2, the cyphertext to be decrypted.
 158:  */
 159: void __send_cyphertext(int32_t *m)
 160: {
 161:     send_instruction(STORE_MESSAGE2);
 162:     send_bits(m, 4);
 163: }
 164:
 165: /*
 166:  * Send data to encrypt/decrypt to device.
 167:  */
 168: void send_int_encrypt_decrypt(int action, int32_t *input, int32_t *output)
 169: {
 170:     if (action == ENCRYPT_SEND)
 171:     {
 172:         send_instruction(STORE_MESSAGE);
 173:         send_bits(input, 4); // cleartext, m
 174:         __read_encryption(output);
 175:     }
 176:
 177:     if (action == DECRYPT_SEND)
 178:     {
 179:         __send_cyphertext(input);
 180:         __read_decryption(output);
 181:     }
 182: }
 183:
 184: /*
 185:  * Return the public keys on this device. Encrypt data already stored
 186:  * on board.
 187:  *
 188:  * (Note: the interface to read private keys was intentionally ommitted.
 189:  */
```

```
190: void __read_encryption(int32_t *encryption)
191: {
192:     int32_t valid[5] = {0,0,0,0,0};
193:     int i;
194:     send_instruction(ENCRYPT_BITS);
195:     send_bits(empty, 2);
196:     read_segment(valid, 5);
197:
198:     while (valid[4] == 0)
199:     {
200:         read_segment(valid+4, 1);
201:     }
202:
203:     read_segment(valid, 5);
204:
205:     for (i = 0; i < 5; i++)
206:     {
207:         encryption[i] = valid[i];
208:     }
209: }
210:
211: void __read_decryption(int32_t *decryption)
212: {
213:     int32_t valid[5] = {0, 0, 0, 0, 0};
214:     int i;
215:
216:     send_instruction(DECRYPT_BITS);
217:     send_bits(empty, 2);
218:     read_segment(valid, 5);
219:
220:     while (valid[4] == 0 || valid[4] == 1)
221:     {
222:         read_segment(valid + 4, 1);
223:     }
224:
225:     read_segment(valid, 5);
226:
227:     for (i = 0; i < 5; i++)
228:     {
229:         decryption[i] = valid[i];
230:     }
231:
232: }
233:
234: /*
235:  * Read "size" 32 bit segments into bit output.
236:  */
237: void read_segment(int32_t *bit_output, int size)
238: {
239:     rsa_box_arg_t rsa_userspace_vals;
240:     int i;
241:
242:     if (rsa_box_fd == -1)
243:         set_fd();
244:
245:     for (i = 0; i < size; i++)
246:     {
247:         rsa_userspace_vals.address = BIT_SEGMENTS_READ[i];
248:
249:         if (ioctl(rsa_box_fd, RSA_BOX_READ_DIGIT, &rsa_userspace_vals))
250:         {
251:             perror("ioctl(RSA_BOX_READ_DIGIT) failed");
252:         }
```

```
253:
254:         bit_output[i] = rsa_userspace_vals.data_in;
255: #ifdef PRINTVERBOSE
256:         printf("[sending] %d // %d\n", BIT_SEGMENTS_READ[i], bit_output[i]);
257: #endif
258:     }
259: }
260:
261: /*
262:  * Get the product of p and q.
263:  */
264: void read_our_N(int32_t *n)
265: {
266:     send_instruction(MAKE_OUR_N);
267:     send_bits(empty, 1);
268:
269:     send_instruction(READ_OUR_N);
270:     send_bits(empty, 1);
271:     read_segment(n, 4);
272: }
273:
274: /** Extended Euclid's implementation below **/
275:
276: #include <string.h>
277: #include <sys/wait.h>
278:
279: #define READ_BUF 4096
280:
281: struct IntSet {
282:     int x[4];
283: };
284:
285: void err_sys(char *err) {
286:     perror(err);
287:     exit(1);
288: }
289:
290: void e_euclid(int32_t e, int32_t phi[4], int32_t *d)
291: {
292:     int phi1 = phi[3];
293:     int phi2 = phi[2];
294:     int phi3 = phi[1];
295:     int phi4 = phi[0];
296:
297:     pid_t pid;
298:     int fd[2];
299:
300:     if(pipe(fd) < 0) {
301:         err_sys("pipe error");
302:     }
303:
304:     if((pid = fork()) < 0) {
305:         err_sys("fork error");
306:     }
307:     else if(pid > 0) { // parent
308:         close(fd[1]); // close write end
309:
310:         if(fd[0] != STDIN_FILENO) { // set STDIN
311:             if(dup2(fd[0], STDIN_FILENO) != STDIN_FILENO) {
312:                 err_sys("dup2 error");
313:             }
314:         }
315:
```

```
316:         char buf[READ_BUF];
317:         if(read(STDIN_FILENO, buf, READ_BUF) < 0) {
318:             err_sys("read error");
319:         }
320:
321:         // printf("[received]: %s\n", buf);
322:
323:         struct IntSet my_s;
324:
325:         /* parse buf */
326:         const char s[2] = " ";
327:         char *token = strtok(buf, s);
328:         int curr = 0;
329:
330:         while(token != NULL && curr < 4) {
331:             my_s.x[curr] = atoi(token);
332:             printf("curr: %d, token: %s\n", curr, token);
333:             token = strtok(NULL, s);
334:             curr++;
335:         }
336:
337:         if (waitpid(pid, NULL, 0) < 0)
338:             err_sys("waitpid error");
339:
340:         d[0] = my_s.x[3];
341:         d[1] = my_s.x[2];
342:         d[2] = my_s.x[1];
343:         d[3] = my_s.x[0];
344:     }
345:     else { // child
346:         close(fd[0]); // close read end
347:
348:         if(fd[1] != STDOUT_FILENO) { // set STDOUT
349:             if(dup2(fd[1], STDOUT_FILENO) != STDOUT_FILENO) {
350:                 err_sys("dup2 error");
351:             }
352:         }
353:
354:         char e_s[READ_BUF];
355:
356:         char phi_s[READ_BUF];
357:         char phi2_s[READ_BUF];
358:         char phi3_s[READ_BUF];
359:         char phi4_s[READ_BUF];
360:
361:         snprintf(e_s, READ_BUF, "%d\n", e);
362:
363:         snprintf(phi_s, READ_BUF, "%d\n", phi1);
364:         snprintf(phi2_s, READ_BUF, "%d\n", phi2);
365:         snprintf(phi3_s, READ_BUF, "%d\n", phi3);
366:         snprintf(phi4_s, READ_BUF, "%d\n", phi4);
367:
368:         printf("%s\n", e_s);
369:
370:         printf("%s\n", phi_s);
371:         printf("%s\n", phi2_s);
372:         printf("%s\n", phi3_s);
373:         printf("%s\n", phi4_s);
374:
375:         // execute Python script
376:         execlp("python", "python", "exteuc.py", e_s, phi_s, phi2_s, phi3_s, phi4_s, (ch
ar *)NULL);
377:     }
```

```
}

```

```c
#include <stdio.h>
#include "instructions.h"

void log_instruction(int opcode)
{
    printf("[sending instruction] ");
    switch(opcode)
    {
        case RESET:
            printf("RESET");
            break;
        case STORE_PUBLIC_KEY_1:
            printf("STORE_PUBLIC_KEY_1");
            break;
        case STORE_PUBLIC_KEY_2:
            printf("STORE_PUBLIC_KEY_2");
            break;
        case DECRYPT_BITS:
            printf("DECRYPT_BITS");
            break;
        case ENCRYPT_BITS:
            printf("ENCRYPT_BITS");
            break;
        case READ_PUBLIC_KEY_1:
            printf("READ_PUBLIC_KEY_1");
            break;
        case READ_PUBLIC_KEY_2:
            printf("READ_PUBLIC_KEY_2");
            break;
        case STORE_MESSAGE:
            printf("STORE_MESSAGE");
            break;
        case STORE_MESSAGE2:
            printf("STORE_MESSAGE2");
            break;
        case STORE_D:
            printf("STORE_D");
            break;
        default:
            break;
    }
    printf("\n");
}
```

```
 1: #include <stdlib.h>
 2: #include "c-interface.h"
 3:
 4: int main(int argc, char **argv)
 5: {
 6:     int32_t p[2];
 7:     int32_t q[2];
 8:
 9:     p[0] = 2;
10:     p[1] = 3;
11:     q[0] = 39;
12:     q[1] = 5000;
13:
14:     set_private_keys(p, q);
15:
16:     return 0;
17: }
```

```
#include <stdio.h>
#include <stdlib.h>

#include <stdint.h>
#include <inttypes.h>

#include <time.h>

static uint64_t primes[50] = {
    (((uint64_t) 1) << 63) - 25,
    (((uint64_t) 1) << 63) - 165,
    (((uint64_t) 1) << 63) - 259,
    (((uint64_t) 1) << 63) - 301,
    (((uint64_t) 1) << 63) - 375,
    (((uint64_t) 1) << 63) - 387,
    (((uint64_t) 1) << 63) - 391,
    (((uint64_t) 1) << 63) - 409,
    (((uint64_t) 1) << 63) - 457,
    (((uint64_t) 1) << 63) - 471,

    (((uint64_t) 1) << 62) - 57,
    (((uint64_t) 1) << 62) - 87,
    (((uint64_t) 1) << 62) - 117,
    (((uint64_t) 1) << 62) - 143,
    (((uint64_t) 1) << 62) - 153,
    (((uint64_t) 1) << 62) - 167,
    (((uint64_t) 1) << 62) - 171,
    (((uint64_t) 1) << 62) - 195,
    (((uint64_t) 1) << 62) - 203,
    (((uint64_t) 1) << 62) - 273,

    (((uint64_t) 1) << 61) - 1,
    (((uint64_t) 1) << 61) - 31,
    (((uint64_t) 1) << 61) - 45,
    (((uint64_t) 1) << 61) - 229,
    (((uint64_t) 1) << 61) - 259,
    (((uint64_t) 1) << 61) - 283,
    (((uint64_t) 1) << 61) - 339,
    (((uint64_t) 1) << 61) - 391,
    (((uint64_t) 1) << 61) - 403,
    (((uint64_t) 1) << 61) - 465,

    (((uint64_t) 1) << 60) - 93,
    (((uint64_t) 1) << 60) - 107,
    (((uint64_t) 1) << 60) - 173,
    (((uint64_t) 1) << 60) - 179,
    (((uint64_t) 1) << 60) - 257,
    (((uint64_t) 1) << 60) - 279,
    (((uint64_t) 1) << 60) - 369,
    (((uint64_t) 1) << 60) - 395,
    (((uint64_t) 1) << 60) - 399,
    (((uint64_t) 1) << 60) - 453,

    (((uint64_t) 1) << 59) - 55,
    (((uint64_t) 1) << 59) - 99,
    (((uint64_t) 1) << 59) - 225,
    (((uint64_t) 1) << 59) - 427,
    (((uint64_t) 1) << 59) - 517,
    (((uint64_t) 1) << 59) - 607,
    (((uint64_t) 1) << 59) - 649,
    (((uint64_t) 1) << 59) - 687,
    (((uint64_t) 1) << 59) - 861,
    (((uint64_t) 1) << 59) - 871
```

```
};

/* GNU C seeder: measures total pseudo-cycles since device on */
unsigned long long rdtsc(){
    unsigned int lo,hi;
    __asm__ __volatile__ ("rdtsc" : "=a" (lo), "=d" (hi));
    return ((unsigned long long)hi << 32) | lo;
}

/* modular exponentiation (base ^ exponent % mod) */
uint64_t modulo(uint64_t base, uint64_t exponent, uint64_t mod) {
    uint64_t x = 1; uint64_t y = base;

    while (exponent > 0) {
        if (exponent % 2 == 1) // odd exponents
            x = (x * y) % mod;
        y = (y * y) % mod;
        exponent = exponent / 2;
    }
    return x % mod;
}

/*
 * Miller-Rabin Primality Test, iteration = accuracy
 */
int miller(uint64_t p, int iteration) {
    int i;
    printf("%" PRIu64 "\n",p);

    if (p < 2) { return 0; }
    if (p != 2 && p % 2 == 0) { return 0; }

    uint64_t s = p - 1;
    while (s % 2 == 0) { s /= 2; }

    for (i = 0; i < iteration; i++) {

        uint64_t a = rand() % (p - 1) + 1, temp = s;
        uint64_t mod = modulo(a, temp, p);

        while (temp != p - 1 && mod != 1 && mod != p - 1) {
            mod = (mod * mod) % p;
            mod = (mod * mod) % p;
            temp *= 2;
        }

        if (mod != p - 1 && temp % 2 == 0) { return 0; }
    }
    return 1;
}

uint64_t get_random(int tries) {

    uint64_t r30 = (uint64_t)rand();    // top 30
    uint64_t s30 = (uint64_t)rand();    // middle 30
    uint64_t t4  = rand() & 0xf;                // bottom 4

    uint64_t res = (r30 << 34) + (s30 << 4) + t4;

    while(tries > 0) {
        if(res > (((uint64_t) 1) << 50))
            res >>= 1;
        tries--;
```

```
127:     }
128:
129:     return res;
130: }
131:
132: uint64_t generate_prime() {
133:
134:     int iteration = 5;
135:     int tries = 0; /* LINEAR BACKOFF */
136:
137:     srand(rdtsc()); // randomize seed
138:
139:     for(;;) {
140:
141:         uint64_t num = 0x0LL; int j;
142:         for(j = 0; j <= tries && j <= 1000; j++) {
143:             // printf("tries: %d, j: %d\n", tries, j);
144:             num = get_random(tries);
145:
146:             if(miller(num, iteration) == 1) { return num; }
147:         }
148:         tries++;
149:     }
150:     return -1;
151: }
152:
153: uint64_t pick_prime() {
154:     int i;
155:     srand(rdtsc());
156:     i = rand() % 50;
157:     return primes[i];
158: }
159:
160:
161:
162: #define BIT_MASK 0xffffffff
163:
164: void generate_prime_as_int32_t(int32_t *prime_64)
165: {
166:     uint64_t prime = pick_prime();
167:     printf("%llX\n", prime);
168:     prime_64[0] = (int32_t) (prime % BIT_MASK);
169:     prime_64[1] = (int32_t) (prime >> 32);
170: }
171:
172: int main() {
173: //    printf("%" PRIu64 "\n", generate_prime());
174: //    printf("%" PRIu64 "\n", generate_prime());
175:     int32_t prime_64[2];
176:     generate_prime_as_int32_t(prime_64);
177:     printf("%X %X", prime_64[0], prime_64[1]);
178:
179:     return 0;
180: }
```

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <time.h>       /* for sleep() */
#include <stdint.h>     /* for unit32_t */
#include "../rsa_box.h"
#include "c-wrapper.h"
#include "instructions.h"

#include "prime-generator.h"

int rsa_box_fd;

// print out 128 bit int, but by [sections]
void print_128_bit_integer(int32_t *input_x)
{
    int i;

    for (i = 0; i < 4; i++)
        printf("quartile(%d): %u\n", i, input_x[i]);
}

/*
 * Return 1 if all size 32 bit numbers in the value are
 * equal; else return 0.
 */
int large_numbers_equal(int32_t *a, int32_t *b, int size)
{
    int i;
    for (i = 0; i < size; i++)
        if (a[i] != b[i]) return 0;

    return 1;
}

int main()
{
    /*
     * main tests
     */
    int32_t p[2];
    int32_t q[2];

    //int32_t n[4];
    int32_t n_our[4]; // our copy of n
    int32_t e_message[4];
    int32_t d_message[4];

    int32_t message[4]  = {13,0,0,0};

    printf("RSA Box device driver started\n");

    /* STORING PRIVATE KEYS, e.g. 23 and 17. */
    p[0] = 23;
    p[1] = 0;
    q[0] = 17;
    q[1] = 0;

```

```
64:     printf("\n[test case: storing private key...]\n");
65:     key_swap(p, q, n_our);
66:
67:     printf("current value of n:");
68:     print_128_bit_integer(n_our);
69:
70:     /* ENCRYPT/DECRYPT TEST */
71:     printf("Original message:");
72:     print_128_bit_integer(message);
73:
74:     send_int_encrypt_decrypt(ENCRYPT_SEND, message, e_message);
75:
76:     printf("Encrypted message:");
77:     print_128_bit_integer(e_message);
78:
79:     send_int_encrypt_decrypt(DECRYPT_SEND, e_message, d_message);
80:
81:     printf("Decrypted message (should match original):");
82:     print_128_bit_integer(d_message);
83:
84:
85:     return 0;
86: }
```